UCRL-91753
PREPRINT

Programming the Livermore VAX 11/780-4 Parallel Processor System Version 1

Edwin Hastings
Nancy Werner

Computing Research Group
Lawrence Livermore National Laboratory

DEC Parallel Processing Conference
Laconia, New Hampshire

November 13-16, 1984

Lawrence Livermore Laboratory

Programming the

Livermore VAX 11/780-4

Parallel Processor System[*]

Version 1

Edwin Hastings
Nancy Werner

Computing Research Group
Lawrence Livermore National Laboratory

November 8, 1984

ABSTRACT. This manual describes the Para Subroutine Library and Utility Package. The Para library is used to program the four processor shared memory VAX 11/780-4. A discussion of the library, utilities, and a demonstration program are provided.

[*]This work was supported by the U.S. Department of Energy under Lawrence Livermore National Laboratory contract # W-7405-ENG-48.

CONTENTS

CHAPTER 1 INTRODUCTION

1.1 MAILBOXES 1-2
1.2 GLOBAL SECTIONS 1-2
1.3 EVENT FLAGS 1-2
1.4 LOCKS 1-3
1.5 BINARY SEMAPHORES (LATCHES) 1-3
1.6 COUNTING SEMAPHORES (P & V) 1-4

CHAPTER 2 THE PARA SUBROUTINES

CHAPTER 3 THE SHARE PROGRAM

CHAPTER 4 RUNNING A PARALLEL PROGRAM

4.1 DESCRIPTION OF COMMANDS 4-5
4.1.1 Newfile 4-5
4.1.2 Paralink 4-5
4.1.3 ParalinkD 4-5
4.1.4 Parasetup 4-5
4.1.5 Parasubmit 4-5
4.1.6 Removeglobal 4-5

APPENDIX A GLOSSARY

November 8, 1984

CHAPTER 1

INTRODUCTION

The Para Libary are the first routines used to program the four processor shared memory Livermore VAX 11/780-4. As a first set they have a few rough edges and need enhancement. These routines are intended to make parallel programming on the VAX straight forward and easy to use.

There is also a listing and description of the share program which was used to debug and test these routines. Share provides and example of global sections, locks, and mailboxes. It simulates a partial differential solution solver. The actual code in the "work" area of share tests the scheduling algorithms.

Five types of routines and their uses are described in this manual.

1.	Mailboxes	- Scheduling
2.	Global Sections	- Shared Variables
3.	Event Flags	- Barriers
4.	Locks	- Critical Sections
5.	Binary Semaphores (Latches)	
6.	Counting Semaphores (P & V)	

If you have suggestions or comments please talk to us.

Edwin Hastings (415) 423-7875 & Nancy Werner (415) 423-7874

1.1 MAILBOXES

Mailboxes are useful for scheduling work among multiple processes. One process writes information about work to be done to the mailbox and any cooperating process can read it.

Mailboxes are sequential devices that reside in memory. They are used the same way that other devices are, with normal READ and WRITE statements. If there are more writes than reads then the data is buffered, if there are more reads than writes then the reading processes are put into a queue to wait for data.

1.2 GLOBAL SECTIONS

Global sections are most useful for putting variables into the shared memory. This is done by placing COMMON blocks into shared memory. When the common block is in shared memory the variables in the common block automatically become shared. This way shared variables are used just like other variables.

Global sections are named areas of memory. The global section routines create sections that are in shared memory so that processes on all machines can access them. You can do this by forcing a common block onto a page boundary (this is done by Paralink) and by using the create and map global section call in your program's initialization section. At the end of your program the global sections must be deleted.

1.3 EVENT FLAGS

Event flags sychronize processes so that they all start or stop at the same point. For example all processes wait on an event flag that says everyone has finished initialization or a process can check an event flag periodically to see if any other process has caused an error or finished its job. Event flags are barriers that activate all the jobs which are waiting on them.

There are 32 event flags that can be shared between processes. Three functions can be performed on event flags. turning them on, turning them off, and waiting for them to be turned on.

An example with 5 processes: process 1 turns off a flag, after a while process 2 waits for the flag, then processes 3, 4, and 5 join in the wait. Finally process 1 decides to turn on the flag. At that time processes 2, 3, 4, and 5 continue execution. All other processes that come along and wait on the flag continue execution because the flag is still on. Later any process may turn off the event flag, but this won't affect the running processes.

1.4 LOCKS

Locks, Counting semaphores, and Binary semaphores are used for protecting critical variables. These are variables that you want to have only one process modify at one time. Locks and semaphores allow you to protect a single variable access or a whole section of code.

Locks, Counting semaphores and Binary semaphores are very similar. They differ in that processes waiting for a lock or counting semaphore wait in a queue and the first process in the queue gets the lock when it is released. With binary semaphores any one of the waiting processes may get it. With locks only the processes that turned it on can turn it off, with semaphores any process can turn it off once it is on. Locks are slower than semaphores but they provide more security and the ability to be defined as hierarchy of locks. There are 32 binary semaphores and an unlimited number of locks and counting semaphores available.

An example with 5 processes. process 1 obtains a lock, after a while process 2 tries to obtain the lock and waits for it since the lock is in use. Then processes 3, 4, and 5 join in the wait. Finally process 1 decides to release the lock. At that time processes 2 continues execution having obtained the lock, and process 3, 4, and 5 continue to wait. All other processes that come along and tries to obtain the lock wait in the queue. Later process 2 turns off the lock and the next process in the queue gets it.

1.5 BINARY SEMAPHORES (LATCHES)

Latches are binary semaphores that are granted to one of the processes that is waiting for them. It isn't possible to determine the order in which processes will get the latch or how long it will take to obtain. There are 32 latches available to the user.

An example with 5 processes: process 1 turns on a latch, after a while process 2 tries to obtain the latch and waits for it since the latch is in use. Then processes 3, 4, and 5 join in the wait. Finally process 1 decides to turn off the latch. At that time one of the processes, either 2, 3, 4, or 5 obtains the latch and continues execution. All other processes that come along and try to obtain the latch wait in the pile.

1.6 COUNTING SEMAPHORES (P & V)

Counting semaphores are similar to Latches (Binary Semaphores) except that they have a queue of waiting processes. This guarantees that once you wait for a semaphore you are in a fixed line for it With binary semaphores anyone who is waiting for it may get it randomly thus someone who starts waiting after you may get it before you do.

An example with 5 processes: process 1 obtains a counting semaphore. after a while process 2 tries to obtain the semaphore and waits for it since the semaphore is in use. Then processes 3, 4. and 5 join in the wait. Finally process 1 decides to relinquish the semaphore. At that time processes 2 continues execution having obtained the semaphore. and process 3. 4. and 5 continue to wait. All other processes that come along and try to obtain the semaphore wait in the queue. Later process 2 relinquishes the semaphore and the next process in the queue gets it.

CHAPTER 2

THE PARA SUBROUTINES

PARA_EVENTS_INIT --- Initialize Event Flags

Para_events_init associates a process with a common event flag cluster. Calling this from multiple processes lets them all use the same event flags provided that the same events_name is specified. This need be done only once per process no matter how many event flags are used.

FORMAT

```
status = para_events_init (events_name)
```

RETURNS

type.	integer*4 status condition
access.	write only
mechanism.	by value

ARGUMENTS

EVENTS_NAME

type.	character
access.	read only
mechanism.	by descriptor

PARA_EVENT_ON —— Turn on an Event Flag

Para_event_on sets an event flag This causes jobs which are waiting on the event flag to continue execution.

FORMAT

```
status = para_event_on (event_number)
```

RETURNS

type.	integer*4 status condition
access	write only
mechanism.	by value

ARGUMENTS

EVENT_NUMBER

type.	integer*4 range 0..31
access.	read only
mechanism.	by reference

PARA_EVENT_OFF —— Turn off an Event Flag

Para_event_off resets an event flag.

FORMAT

```
status = para_event_off (event_number)
```

RETURNS

type	integer*4 status condition
access	write only
mechanism	by value

ARGUMENTS

EVENT_NUMBER

type	integer*4 range 0..31
access	readonly
mechanism	by reference

PARA_EVENT_WAIT —— Wait for an Event Flag

If the event flag is on when PARA_EVENT_WAIT is called then the process continues execution. If the event flag is off when PARA_EVENT_WAIT is called then PARA_EVENT_WAIT waits for the event flag to be turned on before continuing.

FORMAT

status = para_event_wait (event_number)

RETURNS

type.	integer*4 status condition
access	write only
mechanism	by value

ARGUMENTS

EVENT_NUMBER

type:	integer*4 range 0..31
access.	readonly
mechanism	by reference

PARA_GLOBAL_MAP —— Map a Global Section

Para_global_map maps data from first_item to last_item into global section GLOBAL_NAME. This can be used to put data structures and common blocks into the shared memory.

FORMAT

```
status = para_global_map (global_name, first_item_address,
                          last_item_address, filename)
```

RETURNS

type.	integer*4 status condition
access.	write only
mechanism	by value

ARGUMENTS

GLOBAL_NAME

type.	character
access:	read only
mechanism	by descriptor

FIRST_ITEM_ADDRESS

type.	integer*4
access.	read only
mechanism.	by reference

LAST_ITEM_ADDRESS

type:	integer*4
access:	read only
mechanism.	by reference

FILENAME

type.	character
access.	read only
mechanism.	by descriptor
description	This file is used to save the contents of the global section by PARA_GLOBAL_WRITE. This filemust be at least the size of the global section. See the NEWFILE command.

PARA_GLOBAL_MAP_ZRO —— Map and Zero a Global Section

Para_global_map_zro maps zeroed data for first_item to last_item onto global section GLOBAL_NAME This can be used for putting data into shared memory and initializing it to zero.

FORMAT

```
status = para_global_map_zro (global_name,
                              first_item_address,
                              last_item_address, filename)
```

RETURNS

type.	integer*4 status condition
access.	write only
mechanism.	by value

ARGUMENTS

GLOBAL_NAME

type.	character
access:	read only
mechanism:	by descriptor

FIRST_ITEM_ADDRESS

type.	integer*4
access.	read only
mechanism:	by reference

LAST_ITEM_ADDRESS

type.	integer*4
access:	read only

mechanism.	by reference

FILENAME

type	character
access.	read only
mechanism.	by descriptor
description.	This file is used to save the contents of the global section by PARA_GLOBAL_WRITE. This filemust be at least the size of the global section. See the NEWFILE command.

PARA_GLOBAL_REMOVE --- Remove a Global Section

Para_global_remove deletes a global section from memory Global_name is the section to be deleted.

NOTE. Only the processor which creates the global section can remove it from the shared memory. If a non creator processor tries to delete the section then the error status SS$_NOTCREATOR is returned. If the global section does not exists or has already been deleted then the status SS$_NOSUCHSEC is returned.

FORMAT

```
status = para_global_remove (global_name)
```

RETURNS

type:	integer*4 status condition
access.	write only
mechanism.	by value

ARGUMENTS

GLOBAL_NAME

type:	character
access.	read only
mechanism.	by reference

PARA_GLOBAL_WRT — Write a Global Section to its Disk File

Para_global_wrt writes a global section to its corresponding backup file. that is the filename which was passed to the global section map routine. Only the processor which created the section can write it to disk.

FORMAT

```
status = para_global_wrt (first_item_address.
                          last_item_address)
```

RETURNS

type:	integer*4 status condition
access.	write only
mechanism.	by value

ARGUMENTS

FIRST_ITEM_ADDRESS

type:	integer*4
access:	read only
mechanism.	by reference

LAST_ITEM_ADDRESS

type:	integer*4
access.	read only
mechanism:	by reference

PARA_LATCHES_INIT —— Initialize Latches (Binary Semaphores)

Para_Latches_init associates a process with a common event flag cluster. Calling this from multiple processes lets them all use the same Latches provided that the same Latch_name is specified. Latches use event flag clusters in a different way than the Para_event routines do.

NOTE: Locks, Counting semaphores and Binary semaphores are very similar. They differ in that processes waiting for a lock or counting semaphore wait in a queue and the first process in the queue gets the lock when it is released. With binary semaphores any one of the waiting processes may get it. With locks only the processes that turned it on can turn it off, with semaphores any processes can turn it off once it is on. Locks are slower than semaphores but they provide more security and the ability to be defined as hierarchy of locks. There are 32 binary semaphores and an unlimited number of locks and counting semaphores available.

FORMAT

```
status = para_latches_init (latch_name)
```

RETURNS

type:	integer*4 status condition
access:	write only
mechanism.	by value

ARGUMENTS

LATCH_NAME

type:	character
access:	read only
mechanism:	by descriptor

PARA_LATCH_INIT — Initialize a Latch (Binary Semaphore)

Para_latch_init initializes a latch to be available

FORMAT

```
status = para_latch_init (latch_value.latch_number)
```

RETURNS

type.	integer*4 status condition
access.	write only
mechanism	by value

ARGUMENTS

LATCH_VALUE

type.	integer*2 range (<= 1)
access:	read/write
mechanism:	by reference
description.	Latch_value contains the status value of the latch. Status value = 1 means latch is available, <= 0 means latch is in use.

LATCH_NUMBER

type:	integer*2 range 0..31
access.	read only
mechanism.	by reference
description:	Latch_number contains the event flag number which is used to signal the availablility of the latch.

PARA_LATCH_ON —— Obtain a Latch (Binary Semaphore)

Para_latch_on waits for a latch. There may be multiple processes waiting for the latch. When it becomes available only one of the processes will get the latch.

FORMAT

```
status = para_latch_on (latch_value, latch_number)
```

RETURNS

type:	integer*4 status condition
access:	write only
mechanism:	by value

ARGUMENTS

LATCH_VALUE

type:	integer*2 range (<= 1)
access:	read/write
mechanism:	by reference
description:	
	Latch_value contains the status value of the latch. Status value = 1 means latch is available, <= 0 means latch is in use.

LATCH_NUMBER

type:	integer*2 range 0..31
access:	read only
mechanism:	by reference
description:	
	Latch_number contains the event flag number which is used to signal the availablility of the latch.

PARA_LATCH_OFF —— Clear a Latch (Binary Semaphore)

Para_latch_off clears a latch thus making it available to other processes.

FORMAT

```
status = para_latch_off (latch_value, latch_number)
```

RETURNS

type:	integer*4 status condition
access:	write only
mechanism:	by value

ARGUMENTS

LATCH_VALUE

type:	integer*2 range (<= 1)
access:	read/write
mechanism:	by reference
description:	

Latch_value contains the status value of the latch. Status value = 1 means latch is available, <= 0 means latch is in use.

LATCH_NUMBER

type:	integer*2 range 0..31
access:	readonly
mechanism:	by reference
description:	

Latch_number contains the event flag number which is used to signal the availablility of the latch.

PARA_LOCK_ASGN —— Assign Null Lock

Para_lock_asgn creates a null lock called LOCK_NAME Named locks can be used by multiple programs to synchronize access to a certain area of code or variables. The returned value of LOCK_NAME_ID contains a lock identification, which should be used in PARA calls to release or convert the lock, and a status code indicating whether or not the Lock operation was performed. The first word of LOCK_NAME_ID is the status and should be checked after the call for SS$_NORMAL.

NOTE. Locks, Counting semaphores and Binary semaphores are very similar. They differ in that processes waiting for a lock or counting semaphore wait in a queue and the first process in the queue gets the lock when it is released. With binary semaphores any one of the waiting processes may get it. With locks only the processes that turned it on can turn it off, with semaphores any processes can turn it off once it is on. Locks are slower than semaphores but they provide more security and the ability to be defined as hierarchy of locks. There are 32 binary semaphores and an unlimited number of locks and counting semaphores available.

FORMAT

```
status = para_lock_asgn (lock_name,lock_name_id)
```

RETURNS

type.	integer*4 status condition
access:	write only
mechanism:	by value

ARGUMENTS

LOCK_NAME

type.	character
access.	read only
mechanism.	by descriptor
description.	Lock_name is the name of the lock to be used.

LOCK_NAME_ID

type:	integer*2 (4)
access:	write only
mechanism.	by reference
description.	lock_name_id contains the id code needed to refer to the lock on all subsequent para_lock calls. The first word of lock_name_id contains the status code returned by this function.

PARA_SUBLOCK_ASGN --- Null Sublock

Para_sublock_asgn creates a null sublock called LOCK_NAME. Named locks can be used by multiple programs to synchronize access to a certain area of code or variable, a sublock is associated with its parent lock. All sublocks with the same parent lock can be turned off with a single system call. The returned value of LOCK_NAME_ID contains a lock identification, which should be used in PARA calls to release or convert the lock, and a status code indicating whether or not the Lock operation was performed. The first word of LOCK_NAME_ID is the status and should be checked after the call for SS$_NORMAL.

FORMAT

```
status = para_sublock_asgn (lock_name:parent_name_id,
                            lock_name_id)
```

RETURNS

type.	integer*4 status condition
access.	write only
mechanism.	by value

ARGUMENTS

LOCK_NAME

type:	character
access:	read only
mechanism:	by descriptor
description:	Lock_name is the name of the lock to be used.

PARENT_NAME_ID

type:	integer*2 (4)

access.	read only
mechanism.	by reference
description:	
	PARENT_NAME_ID contains the ID. code of the parent lock.

LOCK_NAME_ID

type:	integer*2 (4)
access:	write only
mechanism.	by reference
description.	
	lock_name_id contains the id code needed to refer to the lock on all subsequent para_lock calls. The first word of lock_name_id contains the status code returned by this function.

PARA_LOCK_ON ---- Protected Write Lock

Para_lock_on converts a null lock with value lock_name_id to a protected write lock. Lock_name_id was returned by Para_lock_asgn. If the lock is already on then the process is put into a waiting queue for the lock. Named locks can be used by multiple programs to synchronize access to a certain area of code or variable. The first word of LOCK_NAME_ID is the status and should be checked after the call for SS$_NORMAL.

FORMAT

```
status = para_lock_on (lock_name_id)
```

RETURNS

type:	integer*4 status condition
access:	write only
mechanism:	by value

ARGUMENTS

LOCK_NAME_ID

type:	integer*2 (4)
access:	read/write
mechanism:	by reference
description:	lock_name_id contains the id code needed used to refer to the lock. The first word of lock_name_id contains the status code returned by this function.

PARA_LOCKV_ON —— Protected Write Lock

Para_lockv_on converts a null lock with value lock_name_id to a protected write lock. Lock_name_id was returned by Para_lock_asgn. If the lock is already on then the process is put into a waiting queue for the lock. Para_lockv_on will also retrieve 8 words of data from the distributed lock manager's database into words 5 through 12 of lock_name_id. This data had been previously stored there by a call to Para_lockv_off. The first word of LOCK_NAME_ID is the status and should be checked after the call for SS$_NORMAL.

FORMAT

```
status = para_lockv_on (lock_name_id)
```

RETURNS

type:	integer*4 status condition
access:	write only
mechanism:	by value

ARGUMENTS

LOCK_NAME_ID

type:	integer*2 (12)
access:	read/write
mechanism:	by reference
description:	lock_name_id contains the id code needed used to refer to the lock. The first word of lock_name_id contains the status code returned by this function.

PARA_LOCK_OFF —— Null lock

Para_lock_off converts the lock referred to by lock_name_id to a null lock. Lock_name_id was returned by Para_lock_asgn. If processes are waiting for the lock when it is turned off, then the first process in the queue waiting for the lock will get it.

FORMAT

```
status = para_lock_off (lock_name_id)
```

RETURNS

type.	integer*4 status condition
access.	write only
mechanism.	by value

ARGUMENTS

LOCK_NAME_ID

type:	integer*2 (4)
access:	read/write
mechanism:	by reference
description:	lock_name_id contains the id code needed used to refer to the lock. The first word of lock_name_id contains the status code returned by this function.

PARA_LOCKV_OFF —— Null lock

Para_lock_off converts the lock referred to by lock_name_id to a null lock Lock_name_id was returned by Para_lock_asgn. If processes are waiting for the lock when it is turned off. then the first process in the queue waiting for the lock will get it. Paralockvoff will also store 8 words of data contained in words 5 through 12 of lock_name_id into the distributed lock manager's database.

FORMAT

```
status = para_lockv_off (lock_name_id)
```

RETURNS

type.	integer*4 status condition
access.	write only
mechanism.	by value

ARGUMENTS

LOCK_NAME_ID

type.	integer*2 (12)
access:	read/write
mechanism.	by reference
description.	lock_name_id contains the id code needed used to refer to the lock. The first word of lock_name_id contains the status code returned by this function.

PARA_SUBLOCK_OFF —— Null locks

Para_sublock_off converts all the sublocks of a parent lock to null locks

FORMAT

```
status = para_sublock_off (parent_name_id)
```

RETURNS

type.	integer*4 status condition
access.	write only
mechanism.	by value

ARGUMENTS

PARENT_NAME_ID

type.	integer*2 (4)
access.	read/write
mechanism:	by reference
description:	parent_name_id contains the id code needed used to refer to the parent lock. The first word of parent_name_id contains the status code returned by this function.

PARA_SUBLOCKV_OFF —— Null locks

Para_sublockv_off converts all the sublocks of a parent lock to null locks and stores 8 words of data from words 5 through 12 of parent_name_id into the distributed lock manager's database for each sublock.

FORMAT

status = para_sublockv_off (parent_name_id)

RETURNS

type.	integer*4 status condition
access.	write only
mechanism.	by value

ARGUMENTS

PARENT_NAME_ID

type:	integer*2 (12)
access.	read/write
mechanism.	by reference
description:	parent_name_id contains the id code needed used to refer to the parent lock. The first word of parent_name_id contains the status code returned by this function.

PARA_LOCK_REL —— Release a lock

Para_lock_off releases the lock referred to by lock_name_id. Lock_name_id was returned by Para_lock_asgn.

FORMAT

```
status = para_lock_rel (lock_name_id)
```

RETURNS

type.	integer*4 status condition
access.	write only
mechanism.	by value

ARGUMENTS

LOCK_NAME_ID

type:	integer*2 (4)
access:	read only
mechanism.	by reference
description:	LOCK_NAME_ID contains the ID code of the lock to be released.

PARA_SUBLOCK_REL --- Release sublocks

Para_sublock_rel releases all the sublocks of a parent lock which is referenced by parent_name_id.

FORMAT

```
status = para_sublock_rel (parent_name_id)
```

RETURNS

type.	integer*4 status condition
access.	write only
mechanism.	by value

ARGUMENTS

PARENT_NAME_ID

type:	integer*2 (4)
access:	read only
mechanism:	by reference

PARA_LOCK_REL_ALL —— Release All Locks

Para_lock_rel_all releases all locks held by the current process

FORMAT

```
status = para_lock_rel_all
```

RETURNS

type	integer*4 status condition
access	write only
mechanism	by value

ARGUMENTS

none

PARA_MBX --- Create a Mailbox

Para_mbx opens or creates the mail box MAIL_BOX for the calling process MBX_channel is returned as the logical channel number to read from and write to when using the mailbox.

FORMAT

```
status = para_mbx (mail_box, message_size. mbx_channel)
```

RETURNS

type.	integer*4 status condition
access.	write only
mechanism.	by value

ARGUMENTS

MAIL_BOX

type:	character
access:	read only
mechanism.	by reference

MESSAGE_SIZE

type:	integer*4
access:	read only
mechanism:	by reference

MBX_CHANNEL

type:	integer*4
access:	write only
mechanism:	by reference

PARA_MBX_WRT —— Write a Message to the Mailbox

Para_mbx_wrt sends a message to the mailbox The first process in the read mailbox queue will get the message Message_size is the length (in bytes) of mailbox messages. This is an asynchronous routine.

FORMAT

```
status = para_mbx_wrt (mbx_channel. message_size.
                       mbx_message, qio_status)
```

RETURNS

type	integer*4 status condition
access.	write only
mechanism.	by value

ARGUMENTS

MBX_CHANNEL

type.	integer*4
access.	read only
mechanism.	by reference

MESSAGE_SIZE

type:	integer*4
access.	read only
mechanism.	by reference

MBX_MESSAGE

type.	byte* MESSAGE_SIZE
access.	read only
mechanism.	by reference

QIO_STATUS

type.	integer*2(4)
access.	write only
mechanism	by reference

PARA_MBX_RD —— Read a Message from the Mailbox

Para_mbx_rd reads a message from the mailbox. If there is no message then the process waits in a queue with other processes which are requesting a message from this mailbox. When a message arrives the first process in the queue takes it and continues processing, the next message goes to the next process in the queue, etc. Message_size is the length (in bytes) of mailbox messages.

FORMAT

```
status = para_mbx_rd (mbx_channel, message_size,
                      mbx_message, qio_status)
```

RETURNS

type:	integer*4 status condition
access:	write only
mechanism:	by value

ARGUMENTS

MBX_CHANNEL

type:	integer*4
access:	read only
mechanism.	by reference

MESSAGE_SIZE

type:	integer*4
access:	read only
mechanism:	by reference

MBX_MESSAGE

type:	byte* MESSAGE_SIZE
access:	write only
mechanism:	by reference

QIO_STATUS

type:	integer*2(4)
access:	write only
mechanism:	by reference

PARA_WAKEUP_MAP —— Initialize Wakeup for Counting Semaphores

Para_wakeup_map maps to the system wakeup mailbox This is an initialization routine which is needed for the PARA_V subroutine. PARA_V uses this facility to wakeup jobs on multiple processors. This routine should be called at the beginning of each program that uses PARA_V.

FORMAT

```
status = para_wakeup_map ()
```

RETURNS

type.	integer*4 status condition
access.	write only
mechanism.	by value

PARA_SEMA_INIT — Initialize a Counting Semaphore

Para_sema_init initializes a counting semaphore to ready (available)

FORMAT

```
status = para_sema_init (semaphore, semaphore_queue,
                         max_entries)
```

RETURNS

type	integer*4 status condition
access	write only
mechanism.	by value

ARGUMENTS

SEMAPHORE

type.	integer*4 (2)
access:	read/write
mechanism.	by reference
description.	

Pair of longwords in shared memory. First longword contains the status of the semaphore (1. available, <= 0. in use), second longword points to the associated semaphore queue.

SEMAPHORE_QUEUE

type:	integer*4 (4 * maxentries + 1)
access:	read/write
mechanism.	by reference
description:	

Shared circularly linked list used to queue process PIDs and their nodenames when they are waiting for a semaphore.

MAX_ENTRIES

type:	integer*4
access:	read only
mechanism:	by reference
description:	Maximum number of processes allowed to be waiting for this semaphore at one time.

PARA_P —— Obtain a Counting Semaphore

Para_P obtains the semaphore. If the semaphore is available then this routine returns, else it waits in a queue until the semaphore is available.

NOTE. Locks, Counting semaphores and Binary semaphores are very similar. They differ in that processes waiting for a lock or counting semaphore wait in a queue and the first process in the queue gets the lock when it is released With binary semaphores any one of the waiting processes may get it. With locks only the processes that turned it on can turn it off, with semaphores any processes can turn it off once it is on. Locks are slower than semaphores but they provide more security and the ability to be defined as hierarchy of locks. There are 32 binary semaphores and an unlimited number of locks and counting semaphores available.

FORMAT

```
status = para_p (semaphore)
```

RETURNS

type.	integer*4 status condition
access:	write only
mechanism.	by value

ARGUMENTS

SEMAPHORE

type:	integer*4 (2)
access:	read/write
mechanism.	by reference

PARA_V — Relinquish a Counting Semaphore

Para_V relinquishes a semaphore and returns. If other processes are waiting for the semaphore then the first process on the semaphore queue is woken up.

FORMAT

```
status = para_v(semaphore)
```

RETURNS

type.	integer*4 status condition
access.	write only
mechanism.	by value

ARGUMENTS

SEMAPHORE

type.	integer*4 (2)
access.	read/write
mechanism.	by reference

PARA_FORK —— Submit a Job

Para_fork submits a job to the SYS$BATCH batch queue for execution

---

FORMAT

status = para_fork (process_name. num. [par1,...parNum])

RETURNS

type.	integer*4 status condition on failure batch queue entry number on success
access	write only
mechanism.	by value

ARGUMENTS

PROCESS_NAME

type.	character
access:	read only
mechanism:	by descriptor
description.	name of the command file (without the suffix) to be submitted.

NUM

type.	byte range 0..8
access.	read only
mechanism.	by reference
description.	Number of parameters, PAR1...PARNUM, passed to the batch job.

Par1 ... **PARNUM**

type.	character
access.	read only
mechanism.	by descriptor
description.	passed to batch job

CHAPTER 3

THE SHARE PROGRAM

```
c**********************************************************************
c*                                                                    *
c*                                                                    *
c*                                                                    *
c*                       P R O G R A M   S H A R E                    *
c*                                                                    *
c*                                                                    *
c*                                                                    *
c**********************************************************************
  program share
c
c  Sample Parallel Processing Application
c
c  Developed by :  John Sopka, David Walp, Roger Anderson, Nancy Werner
c             _Edwin Hastings
c
c  Created : 8. 1/84
c
c  Purpose.
c
c This application is intended to run in multiple processes on
c multiple processors concurrently.
c
c Scheduling is via a series of mailbox messages.  Critical
c sections are locked using the system lock manager.  Data
c is shared in the shared memory.
c
c The application is designed to simulate a simple partial
c differential equation solution using a finite difference
c method which propagates in a wave like manner through a
c series of subregions within an array.
c
c Boundary conditions are handled by having only even numbered
c or odd numbered regions computed at one time.
c
c          |----------------|
c          |                | Region 1
c          |----------------|
c          |                | Region 2
c          |----------------|
c          |                | Region 3
c          |----------------|
c          |                | Region 4
c          |----------------|
c          |                |   .
c                               .
c          |----------------|   .
c          |                | Region N
c          |----------------|
c
c
c The calculations will be performed over the entire region
c for time steps of length DeltaT from 0 to Tmaximum.
c
```

```
c VMS system symbol definitions
  include '($SSDEF)'      ! VMS system failure and status codes
c*********************************************************************
c*                                                                   *
c*        Type Definitions for Mathematical Solution                 *
c*                                                                   *
c*********************************************************************
  integer  regionsize, arraysize, Nregions, DeltaT, Tmaximum
  parameter       (regionsize =  10,
  1                arraysize  = 100,
  2                Nregions   = arraysize/regionsize,! # regions
  3                DeltaT     =   1,
  4                Tmaximum   = 100,
  5                efnup = 10)
c Data to be shared by multiple cooperating processes
  integer array
  common /data  array(arraysize,arraysize)
                          ! this is the problem data array which
                          ! should appear in shared memory for
                          ! manipulation by all cooperating
                          ! processes.
  integer donearray, lastregion, lastinterval
  common /sched  donearray(Nregions),! scheduling common block.
  1              lastregion,         ! should be in shared memory
  2              lastinterval,       ! to keep track of work
  3              initdone            ! already done
  common /newpage/ dummy  ! this forces data defined after
                          ! the shared common blocks
                          ! to be in local memory
c*********************************************************************
c*                                                                   *
c*        Type Definitions for Parallel Processing                   *
c*                                                                   *
c*********************************************************************
  integer*2       maxmesgsize      ! for mailbox messages
  character*(*)   mbxname ! between processes
  parameter       (mbxname = 'MAILBOX', maxmesgsize = 20)
  integer paralockasgn,   ! The PARAllel processing routines
  2       paralockon,     ! are functions
  3       paralockoff,  !
  4       paralockrel,    ! A normal return is SS$NORMAL.
  5       parambx,
  6       parambxwrt,
  7       parambxrd,
  8       paraglobalmapzro,
  9       paraglobalremove
  integer  getnodename     ! returns the current network node
  integer datalocation(2), schedlocation(2)
                 ! addresses of data to map to Global Sections
c data local to each process
  integer  status
  integer*2 mbxstatus(4), mbxchan
c Status block for Locks
  integer*2 gscnameid(4),  ! Global SeCtion lock
```

```
   1          initnameid(4), ! INITialize lock
   2          updatenameid(4)! UPDATE lock
   character*15     nodename
c BYTEMSG is the mailbox message shared between processes.
   byte             bytemsg(maxmesgsize), msgtype
   integer          timeinterval, workregion,
   1                timeintervalx, workregionx
   equivalence      (bytemsg(1), timeintervalx),
   1                (bytemsg(5), workregionx),
   2                (bytemsg(9), msgtype)
c*********************************************************************
c*                                                                   *
c*      End of Definitions                                           *
c*                                                                   *
c*********************************************************************
   status = getnodename(%DESCR(nodename))  ! get local node name
   if ( status .ne. ss$normal ) call lib$signal (%val(status))
   print 9988,nodename
c*********************************************************************
c*                                                                   *
c*              ASsiGN all the locks                                 *
c*                                                                   *
c*********************************************************************
   status = paralockasgn('GSCLOCK',gscnameid)
   if (status .ne. ss$normal) call lib$signal (%val(status))
   if (gscnameid(1) .ne. ss$normal)
   1                    call lib$signal(%val(gscnameid(1)))
   status = paralockasgn('INITLOCK',initnameid)
   if (status .ne. ss$normal) call lib$signal (%val(status))
   if (initnameid(1) .ne. ss$normal)
   1                    call lib$signal(%val(initnameid(1)))
   status = paralockasgn('UPDATELOCK',updatenameid)
   if (status .ne. ss$normal) call lib$signal (%val(status))
   if (updatenameid(1) .ne. ss$normal)
   1                    call lib$signal(%val(updatenameid(1)))
c*********************************************************************
c*                                                                   *
c*              Start Critical Region Create Global SeCtions         *
c*                                                                   *
c*********************************************************************
   status = paralockon(gscnameid)! lock critical section or
                                 ! wait for lock to be freed
c Check the status code to make sure that the subroutine executed OK
   if (status .ne. ss$normal) call lib$signal (%val(status))
c Check the status to make sure that we obtained the LOCK.
   if (gscnameid(1) .ne. ss$normal)
   1                   call lib$signal(%val(gscnameid(1)))
c Create global section for common block DATA
   datalocation(1) = %loc(array(1,1))
                         ! get addresses of first and last
                         ! elements of the common block.
                         ! data between these address will
                         ! be put into a global section and
                         ! shared memory
```

```
        datalocation(2) = %loc(array(arraysize,arraysize))+4
        status = paraglobalmapzro('DATASEC',datalocation(1),
        1       datalocation(2),'DATASEC.DAT')
        if ( (status .ne. ss$normal) .and. (status .ne. ss$created) )
        1               call lib$signal(%val(status))
                                ! a return code of "section created" is OK.
c Create global section for common block SCHED
        schedlocation(1) = %loc(donearray(1))
        schedlocation(2) = %loc(initdone)+4
        status = paraglobalmapzro('SCHEDSEC',schedlocation(1),
        1       schedlocation(2),'SCHEDSEC.DAT')
        if ( (status .ne. ss$normal) .and. (status .ne. ss$created) )
        1               call lib$signal (%val(status))
c unlock the initialization lock obtained above
        status = paralockoff(gscnameid)
        if  ( status .ne. ss$normal )  call lib$signal( %val(status) )
c*********************************************************************
c*                                                                   *
c*                  Exit Critical Region Global SeCtion              *
c*                                                                   *
c*********************************************************************
c Create scheduling mailbox
        status = parambx(%descr(mbxname).maxmesgsize,mbxchan)
        if ( status .ne. ss$normal )   call lib$signal (%val(status))
c*********************************************************************
c*                                                                   *
c*                  Start Critical Region Initialize                 *
c*                                                                   *
c*********************************************************************
        status = paralockon(initnameid)! lock critical region or
                                        ! wait for lock to be freed
        if ( status .ne. ss$normal ) call lib$signal (%val(status))
        if ( initnameid(1) .ne. ss$normal )
        1       call lib$signal (%val(initnameid(1)))

        if ( initdone .eq. 0 )  then ! Do the initialization

            lastregion = Nregions    ! Initialize synchronization data
            lastinterval = Tmaximum
            do  10  i = 1, Nregions  ! Set number of regions completed to 0
10                  donearray(i) = 0
            do  20  j = 1, arraysize ! Init. problem data for time 0
              do  20  i = 1, arraysize
20                  array( i, j ) = 0.0
            msgtype=1                   ! Send first message to start
            workregionx = 1             ! the processes
            timeintervalx = 1
c           Send the scheduling message to the mailbox
            status=parambxwrt(mbxchan, maxmesgsize, bytemsg, mbxstatus)
            if  ( status  .ne.  ss$normal ) call lib$signal(%val(status))
            if  ( mbxstatus(1)  .ne.  ss$normal )
    1                   call lib$signal( %val(mbxstatus(1)) )
            initdone = 1 ! Let future processes know that init has been done
        end if
```

```
   status = paralockoff(initnameid) ! Release lock obtained above
   if  ( status .ne. ss$normal )  call lib$signal( %val(status) )
c*********************************************************************
c*                                                                   *
c*                  Exit Critical Region Initialize                  *
c*                                                                   *
c*********************************************************************
c*********************************************************************
c*                                                                   *
c*                                                                   *
c*                  Begin Program's Main Work                        *
c*                                                                   *
c*                                                                   *
c*********************************************************************
100       continue        ! wait for a message containing work to do
c Read scheduling message
   status=parambxrd(mbxchan.maxmesgsize,bytemsg.mbxstatus)
   if (status .ne. ss$normal) call lib$signal(%val(status))
   if (mbxstatus(1) .ne. ss$normal)call lib$signal(%val(mbxstatus(1)))
   if ( msgtype .eq. 2 )   goto 900 ! 2 is an Exit Program Command
   if ( msgtype .eq. 1 )  goto 500 ! 1 is an Work On Region Command
   type *.'%Error — Unknown Command read from scheduling mailbox.'
   call exit(1)
500       continue
   workregion   = workregionx  ! workregionx and timeintervalx were
   timeinterval = timeintervalx! read from the mailbox into record
                               ! byte_msg
c Perform calculation for a time interval on one region of the problem
space
          call work( timeinterval,
   1         (workregion - 1)*regionsize + 1,
   2         workregion * regionsize,
   3         arraysize, arraysize)
c If the entire problem has been computed over the full range of time
then exit
   if  ((workregion .eq. lastregion)
   1    .and. (timeinterval .eq. lastinterval)) goto 900
c*********************************************************************
c*                                                                   *
c*         Enter Critical Section for Updating Scheduling Data       *
c*                                                                   *
c*********************************************************************
   status = paralockon(updatenameid)
   if ( status .ne. ss$normal) call lib$signal(%val(status))
   if  ( updatenameid(1) .ne. ss$normal )
   1        call lib$signal( %val(updatenameid(1)) )
   donearray(workregion) = timeinterval ! we have done workregion for
                                        ! this time step
   if  (( workregion .gt. 2)
   1   .and. (timeinterval .lt. Tmaximum))    then
       if  ( donearray( workregion-2 ) .eq. timeinterval+1 )  then

c         Schedule work for previous region for next time step
          msgtype       = 1
```

```
                 workregionx    = workregion-1
            timeintervalx = timeinterval+1
            status=parambxwrt(mbxchan,maxmesgsize,bytemsg,mbxstatus)
            if  (status  .ne.  ss$normal)  call lib$signal( %val(status) )
            if  ( mbxstatus(1)  .ne.  ss$normal )
     1                call lib$signal( %val(mbxstatus(1)) )
         end if ' work for previous region for next time step
     else if  (( workregion .eq. 2)
     1        .and. (timeinterval .lt. Tmaximum)) then
c        Schedule work for current region for next time step
         msgtype        = 1
              workregionx    = 1
         timeintervalx = timeinterval+1
         status=parambxwrt(mbxchan,maxmesgsize,bytemsg,mbxstatus)
         if  ( status  .ne.  ss$normal )  call lib$signal( %val(status) )
         if  ( mbxstatus(1)  .ne.  ss$normal )
     1                call lib$signal( %val(mbxstatus(1)) )
        end if ! work for current region for next time step
     if  ( workregion .lt. Nregions-1 )    then
         if  ( donearray( workregion+2 ) .eq. timeinterval-1 )    then
c           Schedule work for next region for current time step
            msgtype        = 1
                 workregionx    = workregion+1
            timeintervalx = timeinterval
            status=parambxwrt(mbxchan,maxmesgsize,bytemsg,mbxstatus)
            if  (status  .ne.  ss$normal)  call lib$signal(%val(status))
            if  ( mbxstatus(1)  .ne.  ss$normal )
     1                call lib$signal( %val(mbxstatus(1)) )
         end if ! work for next region for current time step
     else if  ( workregion .eq. Nregions-1 )    then
c        Schedule work for last region current time step
         msgtype        = 1
              workregionx    = Nregions
         timeintervalx = timeinterval
         status=parambxwrt(mbxchan,maxmesgsize,bytemsg,mbxstatus)
         if  ( status  .ne.  ss$normal )  call lib$signal( %val(status) )
         if  ( mbxstatus(1)  .ne.  ss$normal )
     1                call lib$signal( %val(mbxstatus(1)) )
     end if ! work for last region current time step
     status = paralockoff(updatenameid) ! release the critical lock
     if  ( status .ne.  ss$normal ) call lib$signal(%val(status))
c*****************************************************************
c*                                                               *
c*      End of Critical Section for Updating Scheduling Data     *
c*                                                               *
c*****************************************************************
     go to 100
c*****************************************************************
c*                                                               *
c*                                                               *
c*        Exit Routine                                           *
c*                                                               *
c*                                                               *
c*****************************************************************
```

```
900     continue
  msgtype       = 2 ! Write to the mailbox to kill other processes
        workregionx   = 0
  timeintervalx = 0
  status=parambxwrt(mbxchan.maxmesgsize,bytemsg.mbxstatus)
  if  ( status  .ne   ss$normal  )  call lib$signal(%val(status))
  if  ( mbxstatus(1)  .ne.  ss$normal  )
  1              call lib$signal( %val(mbxstatus(1)))
c*********************************************************************
c*                                                                   *
c*      Start of Critical Section for Deleting Global Sections       *
c*                                                                   *
c*********************************************************************
  status = paralockon(gscnameid)! lock critical region or
                                  ! wait for lock to be freed
  status = paraglobalremove('DATASEC') ! remove global section DATASEC
  if (status .eq. ss$notcreator) then
        type *.
  1 '%INFO—Failed to delete Global Section DATASEC: Not creator VAX.'
  else if (status .eq. ss$nosuchsec) then
        type *.
  1 '%INFO—Failed to delete Global Section DATASEC. No such section.'
  else if (status .eq  ss$normal) then
        type *.'%INFO—Global Section DATASEC deleted.'
  else
        call lib$signal (%val(status))
  endif
  status = paraglobalremove('SCHEDSEC')! remove global section SCHEDSEC
  if (status .eq. ss$notcreator) then
        type *,
  1 '%INFO—Failed to delete Global Section SCHEDSEC: Not Creator VAX.'
  else if (status .eq. ss$nosuchsec) then
        type *,
  1 '%INFO—Failed to delete Global Section SCHEDSEC. No such section.'
  else if (status .eq. ss$normal) then
        type *.'%INFO—Global Section SCHEDSEC deleted.'
  else
        call lib$signal (%val(status))
  endif
  status = paralockoff(gscnameid) ! release the critical lock
  if  ( status .ne.  ss$normal ) call lib$signal(%val(status))
c*********************************************************************
c*                                                                   *
c*      End of Critical Section for Deleting Global Sections         *
c*                                                                   *
c*********************************************************************
c*********************************************************************
c*                                                                   *
c*                REL all the locks and end the program              *
c*                                                                   *
c*********************************************************************
  status = paralockrel(gscnameid)
  if (status .ne. ss$normal) call lib$signal (%val(status))
  status = paralockrel(initnameid)
  if (status .ne. ss$normal) call lib$signal (%val(status))
  status = paralockrel(updatenameid)
  if (status .ne. ss$normal) call lib$signal (%val(status))
  print 9989, nodename
  call exit
c Format statements
9000      FORMAT(A4.I8.I8)
9988      FORMAT( ' Share execution started on node ',A15/)
9989      FORMAT( /' Share execution completed on node ',A15/)
  end ! program share
c*********************************************************************
c*                                                                   *
c*                                                                   *
c*                                                                   *
c*                    S U B R O U T I N E   W O R K                  *
c*                                                                   *
c*                                                                   *
c*                                                                   *
c*********************************************************************
          subroutine work(timeinterval, startrow, endrow,
 1        rowlength, lastrow)
c
c     This routine checks the array for validity up to this
c     point.  If the data in region 'region'-1 equals 'timeinterval'
c     and region 'region' and 'region'+1 equal 'timeinterval'-1 then
c     the data is valid and the processes are assumed to be
c     running correctly together.
c
c     If an error is found then the error is written to the
c     file on unit 1.
c
c Parameters
c     Timeinterval - time interval to verify for.
c     Startrow - first row in region to check
c     Endrow - last row in region to check
c     Rowlength - length of each row
c     Lastrow - last row in the array
c
c problem parameters
  integer  regionsize, arraysize, Nregions, DeltaT, Tmaximum
  parameter       (regionsize =  10,
  1                arraysize  = 100,
  2                Nregions   = arraysize/regionsize,
  3                DeltaT     =   1,
  4                Tmaximum   = 100)
  integer array
  common /data/ array(arraysize,arraysize)
                                   ! this is the problem data array which
                                   ! should appear in shared memory for
                                   ! manipulation by all cooperating
                                   ! processes.
  integer donearray, lastregion, lastinterval
  common /sched/  donearray(Nregions),
  1               lastregion,
     2               lastinterval,
     3               initdone
     common /newpage/ dummy  ! this forces data defined after the shared
                             ! common blocks into local memory
     integer regionlen       ! length of the region.
c input parameters
     integer timeinterval.  startrow. endrow. rowlength.    lastrow
     regionlen = endrow - startrow + 1
c check the region for timeinterval - 1
     if ( startrow - regionlen .ge. 1) then
             do 10 i= startrow-regionlen, endrow-regionlen
                 do 20 j= 1. rowlength
                     if ( array(i,j) .ne. timeinterval ) then
                        print 100, startrow, i,j,timeinterval,
     1                                 array(i.j)
                        goto 35
                     endif
20                      continue
10                  continue
     endif
c Check the region for timeinterval now
35          continue
     do 30 i= startrow. endrow
         do 40 j= 1, rowlength
             if ( array(i,j) .ne. (timeinterval - 1) ) then
                print 110. startrow, i,j,timeinterval,
     1                         array(i.j)
                goto 45
             endif
40              continue
30          continue
c Check the region for timeinterval+1
45          continue
     if ( startrow + regionlen .lt. lastrow) then
             do 50 i= startrow+regionlen. endrow+regionlen
                 do 60 j= 1. rowlength
                     if ( array(i,j) .ne. (timeinterval - 1) ) then
                        print 120, startrow, i,j,timeinterval,
     1                                 array(i,j)
                        goto 65
                     endif
60                      continue
50                  continue
     endif
c replace the central rows
65          continue
     do 70 i= startrow, endrow
         do 80 j= 1, rowlength
             array(i,j) = timeinterval
80              continue
70          continue
100         format('%Error - d t-1 : str ',i4,' i ',i4,' j ',i4,
     1          ' timeint ',i4, ' a(i,j) ', i4)
110         format('%Error - d t   : str ',i4,' i ',i4,' j ',i4,
     1        ' timeint ',i4, ' a(i,j) ', i4)
   120        format('%Error - d t+1 : str ',i4,' i ',i4,' j ',i4,
     1        ' timeint ',i4, ' a(i,j) ', i4)
      return
      end
```

CHAPTER 4

RUNNING A PARALLEL PROGRAM

Below is a transcript of a terminal session to setup, compile, link, and run the share program. The steps for doing so are shown below and then described on the following page. All input that was typed by the user is underlined with annotations being surrounded by braces ("}{").

Example 1

```
Username: USER
Password.
         Welcome to VAX/VMS version 4.0 on node VAX1
     Last interactive login on Monday, 15-OCT-1984 14:07
     Last non-interactive login on Thursday, 11-OCT-1984 09:48
$ parasetup
Parasetup Procedure, at 14:08:00

Name of program (process): share
Is this a restart job [N]:.
Enter name of shared memory }ATTIC,LOFT,None{ [ATTIC].:
Enter name of shared Global Section or c/r. DATASEC
Enter name of shared Global Section or c/r: SCHEDSEC
Enter name of shared Global Section or c/r.
Enter name of shared Mailbox or c/r: MAILBOX
Enter name of shared Mailbox or c/r:
Enter name of Shared Common Event Flag Cluster or c/r.

To define shared logicals interactively type. @LOGICALSHARE
To submit your program use the PARASUBMIT command.

Finished at 14:08:21
$ fortran share
$ paralink
Paralink Procedure, started at 14:08:48

Enter process name to link: share
Enter common block name or c/r. data
Enter common block name or c/r: sched
Enter common block name or c/r. newpage
```

```
    Enter common block name or c/r:
    Finished at 14:09:02
    $ type share.map
                                                } Some listing omitted {
                                                +-----------------------------+
                                                ! Program Section Synopsis !
                                                +-----------------------------+

   Psect Name        Module Name        Base      End            Length
   ----------        -----------        ----      ---            ------
   $PDATA                               00000200 0000050B 0000030C (        780.)
                     SHARE              00000200 0000043E 0000023F (        575.)
                     WORK               00000440 00000508 000000C9 (        201.)
   $LOCAL                               00000600 00000A87 00000488 (       1160.)
                     SHARE              00000600 000007BF 000001C0 (        448.)
                     WORK               000007C0 000007CB 0000000C (         12.)
                     GET_NODENAME       000007CC 0000081B 00000050 (         80.)
                     PARA_GLOBAL_MAP_ZRO
                                        0000081C 000008EF 000000D4 (        212.)
                     PARA_GLOBAL_REMOVE
                                        000008F0 0000090B 0000001C (         28.)
                     PARA_LOCK_OFF      0000090C 0000093B 00000030 (         48.)
                     PARA_LOCK_ON       0000093C 0000098F 00000054 (         84 )
                     PARA_MBX           00000990 000009BB 0000002C (         44.)
                     PARA_MBX_RD        000009BC 00000A1F 00000064 (        100.)
                     PARA_MBX_WRT       00000A20 00000A87 00000068 (        104.)
   DATA                                 00000C00 0000A83F 00009C40 (      40000.)
                     SHARE              00000C00 0000A83F 00009C40 (      40000.)
                     WORK               00000C00 0000A83F 00009C40 (      40000.)
   NEWPAGE                              0000AA00 0000AA03 00000004 (          4.)
                     SHARE              0000AA00 0000AA03 00000004 (          4.)
                     WORK               0000AA00 0000AA03 00000004 (          4.)
   SCHED                                0000AC00 0000AC33 00000034 (         52.)
                     SHARE              0000AC00 0000AC33 00000034 (         52.)
                     WORK               0000AC00 0000AC33 00000034 (         52.)
    $ newfile
    Newfile Procedure, started at 14:16.02

    File to create or c/r: datasec.dat
    Size in bytes: 40000 } gotten from DATA in the map listing }
    File DATASEC.DAT created with a size of 79 blocks.

    File to create or c/r: schedsec.dat
    Size in bytes: 52 } gotten from SCHED in the map listing }
    File SCHEDSEC.DAT created with a size of 1 blocks.

    File to create or c/r:
    Finished at 14:16:30
    $ parasubmit
    Parasubmit Procedure, started at 14:09:46

    Name of program: share
    Number of jobs [1]: 4
    Start after [current time 14:09:49]: 14:11
```

```
 Queue [sys$batch].

 To stop running jobs type @SHAREDELETE
 After stopping jobs type REMOVEGLOBAL

 Finished at 14.10 06
 $ shb
 Generic batch queue SYS$BATCH

    Jobname          Username      Entry  Status
    -------          --------      -----  ------
    RUNSHARE         ED              886  Holding until
15-OCT-1984 14.11
    RUNSHARE         ED              887  Holding until
15-OCT-1984 14.11
    RUNSHARE         ED              888  Holding until
15-OCT-1984 14.11
    RUNSHARE         ED              889  Holding until
15-OCT-1984 14.11

 Batch queue VAX1$BATCH, on VAX1..

 Batch queue VAX2$BATCH. on VAX2..

 Batch queue VAX3$BATCH. on VAX3..

 Batch queue VAX4$BATCH, on VAX4:.
 $ show time
   15-OCT-1984 14:10.50
 $ shb
 Generic batch queue SYS$BATCH

 Batch queue VAX1$BATCH. on VAX1::

    Jobname          Username      Entry  Status
    -------          --------      -----  ------
    RUNSHARE         ED              886  Executing

 Batch queue VAX2$BATCH, on VAX2:.

    Jobname          Username      Entry  Status
    -------          --------      -----  ------
    RUNSHARE         ED              887  Executing

 Batch queue VAX3$BATCH, on VAX3:.

    Jobname          Username      Entry  Status
    -------          --------      -----  ------
    RUNSHARE         ED              888  Executing

 Batch queue VAX4$BATCH, on VAX4::

    Jobname          Username      Entry  Status
    -------          --------      -----  ------
   RUNSHARE        ED                 889  Executing
 $ show time
   15-OCT-1984 14:11:17
 $
```

When running the program after the first setup the process name can be given after the paralink to avoid all the questions being asked.

Example 2

```
$ paralink share
Paralink Procedure, started at 15:09:06

Finished at 15.10.12
$
```

4.1 DESCRIPTION OF COMMANDS

4.1.1 Newfile

Newfile creates a file of a given size This is used because global sections need files as a backup disk storage. Find the size for the file in the map listing that is created by the linker. An example is shown about.

4.1.2 Paralink

Paralink links your with the Para Library. If you give a program name on the command line (as shown in example 2) then Paralink uses a file that contains all the options it needs. If you don't give a program name (as shown in example 1) then Paralink will ask you for information it needs to ready common blocks for the shared memory.

4.1.3 ParalinkD

ParalinkD adds the VAX Symbolic Debugger into your program. The instructions for ParalinkD are the same as for Paralink. In order to take full advantage of the debugger you should compile your program with the /DEBUG switch.

4.1.4 Parasetup

Parasetup creates a file used when you run your program interactively or in batch. The created file defines privileges, symbols, and logical names needed to run parallel programs.

4.1.5 Parasubmit

Parasubmit submits your program to the batch queue to run. You must have run Parasetup before running your job.

4.1.6 Removeglobal

Removeglobal removes global sections that may have been left in the shared memory if you aborted your program during debugging.

APPENDIX A

GLOSSARY

CRITICAL REGION

A section of code that is surrounded by locks or latches because multiple processes executing it at the same time would cause indeterminate results. Critical regions must be identified by the programmer and can be locked with the para_lock_assign. para_lock_on. para_lock_off. and para_lock_deassign subroutines or the latch routines. para_latches_init. para_latch_init. para_latch_on. and para_latch_off.

MA780

The DEC shared memory unit controller. The VAX 11/780-4 has two MA780s each with 2mb for a total of 4mb (million bytes) shared memo S

EVENT

Change in process status or the occurrence of some activity that concerns single or multiple processes. Events can either be waited for or they can occur asynchronously.

EVENT FLAG

A bit that can be set or cleared to indicate the occurrence of the event associated with the flag. Event flags are used to synchronize activities in a process or among many processes. Programs can wait and use events with the para subroutines.

EVENT FLAG CLUSTER

A set of 32 event flags used for event posting. Four clusters are defined for each process: two process-local cluster and two common event flag cluster. Of the process-local event flags, eight are reserved for system use. The common event flags can be shared among multiple processes. Para subroutines give the user one common event cluster as event flags, and the second event flag cluster as latches.

GLOSSARY

GLOBAL SECTION

A data structure (e.g., FORTRAN global common) or shareable image section potentially available to all processes in the system. Access is protected by privilege and/or group number of the UIC.

MAILBOX

A software data structure that is treated as a record-oriented device for general interprocess communication. Communication using a mailbox is similar to other forms of device-independent I/O. Senders write to a mailbox, the receiver reads from that mailbox.

MULTIPORT MEMORY

A memory unit that can be connected to multiple processors and that can contain resource (for example, mailboxes, common event flag clusters, and global sections) for use by processes running on different processors.

MUTEX

A semaphore that is used to control exclusive access to a region of code that can share a data structure or other resource. The mutex (mutual exclusion) semaphore ensures that only one process at a time has access to the region of code.

PRIVATE SECTION

An image section of a process that is not shareable among processes. See also global section.

PROCESS

The basic entity scheduled by the system software that provides the context in which an image executes. A process consists of an address space and both hardware and software context.

INDEX

Batch
  see parasubmit

Clean up
  see removeglobal
Commands, 4-5
Critical region, A-1

Debugging
  see paralinkd

Event, A-1
Event flag, A-1
Event flag cluster, A-1
Example
  assign lock, 3-4
  check status, 3-4
  common block, 3-3
  create mailbox, 3-5
  global section, 3-5
  lock off, 3-5
  lock on, 3-4
  node name, 3-4
  read mailbox, 3-6
  release lock, 3-8
  remove global section, 3-8
  write mailbox, 3-5

Global section, A-2

Linking program
  see paralink

Ma780, A-1
Mailbox, A-2
Multiport memory, A-2
Mutex, A-2

Newfile, 4-5

Para_event_off, 2-4
Para_event_on, 2-3
Para_event_wait, 2-5
Para_events_init, 2-2
Para_fork, 2-39
Para_global_map, 2-6
Para_global_map_zro, 2-8
Para_global_remove, 2-10
Para_global_wrt, 2-11
Para_latch_init, 2-13
Para_latch_off, 2-15
Para_latch_on, 2-14
Para_latches_init, 2-12
Para_lock_asgn, 2-16
Para_lock_off, 2-22
Para_lock_on, 2-20
Para_lock_rel, 2-26
Para_lock_rel_all, 2-28
Para_lockv_off, 2-23
Para_lockv_on, 2-21
Para_mbx, 2-29
Para_mbx_rd, 2-32
Para_mbx_wrt, 2-30
Para_p, 2-37
Para_sema_init, 2-35
Para_sublock_asgn, 2-18
Para_sublock_off, 2-24
Para_sublock_rel, 2-27
Para_sublockv_off, 2-25
Para_v, 2-38
Para_wakeup_map, 2-34
Paralink, 4-5
Paralinkd, 4-5
Parasetup, 4-5
Parasubmit, 4-5
Private section, A-2
Process, A-2

Removeglobal, 4-5

Shared common block
  see global section
Shared memory
  see ma780

Overview of the

Livermore VAX 11/780-4

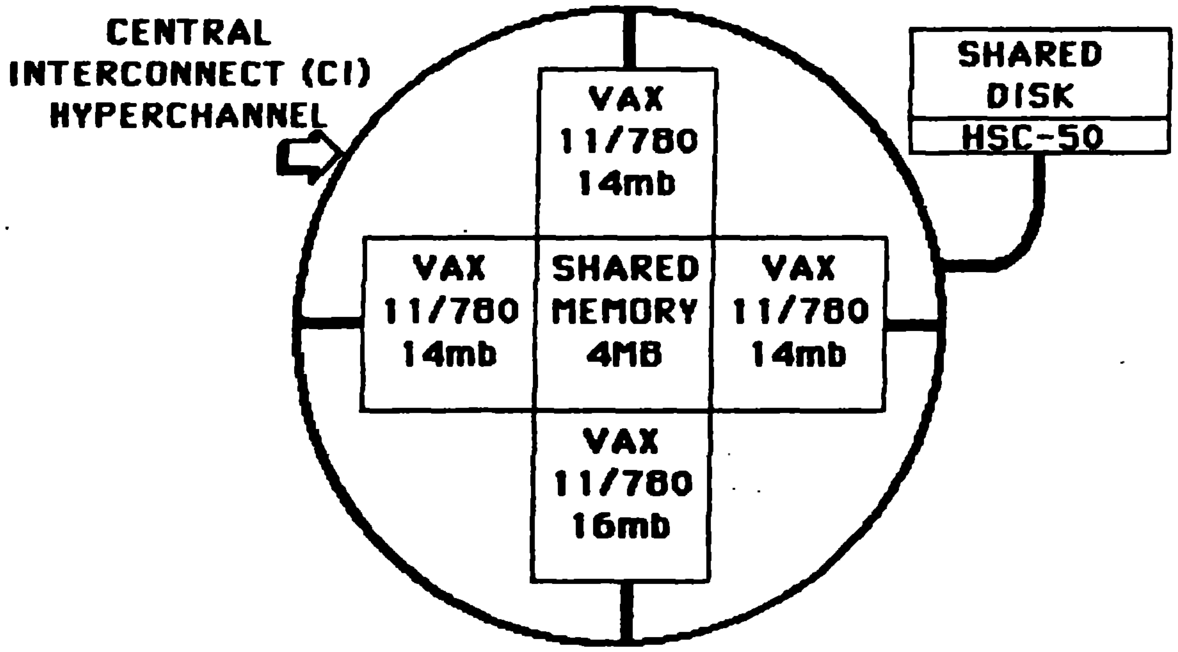
Parallel Processor

Roger Anderson
Edwin Hastings
Nancy Werner

Computing Research Group
Lawrence Livermore National Laboratory

Livermore*para*VAX

Livermore VAX 11/780-4 Hardware



Livermore *para*VAX

VAX 11/780-4 SYSTEM INFORMATION

- Uses Existing Hardware and Software
- Parallel Processor - 4 Vax 11/780 with local and shared memory
- VaxCluster supported under VMS 4.0 provides shared disks and distributed lock manager using 70mb/sec Central Interconnect (CI).
- LLNL Vax 11/780-4 Configuration:
  - 4 Vax 11/780, 14mb memory each
  - 1 MA780 4mb shared memory
  - 1 HSC50 CI disk controller
  - 4 RA80 disks, 125mb each
  - tape drive, line printer, Ethernet
  - VMS 4 FT2 operating system with FORTRAN, Pascal, C, etc.
- Differences with VAX 11/782

Livermore *para*VAX

OUTLINE

1. Why Parallelism
2. VAX 11/780-4 System
3. Collaborators
4. Current Research at LLNL
5. Para Subroutines
6. Hand Coding Parallelism: An Example
7. Conclusion

Livermore*para*VAX

HAND CODING PARALLELISM:

THE SHARE PROGRAM

- Detection of Parallelism
- Granularity
- Speedup
- Stopping Conditions
- Error Handling
- Debugging
- Testability / Reliability

Livermore*para*VAX

CURRENT VAX 11/780-4 RESEARCH at LLNL

- Developing parallel system libraries, the Para subroutines
- Recruiting Vax 11/780-4 users
- Planning Cray COS compatibility
- Parallel demonstration program using shared and local memory
- Divide and Conquer Integration
- Run-Time Algorithm Selection
- SISAL Data Flow Language
- Parallel Algorithm Approach

Livermore *para* VAX

VAX 11/780-4 DEC COLLABORATORS

SITE	STATE	
LLNL	Running	
LBL	Running	[Maples]
Nasa Ames	Running	[Stevens]
CMU	1st Running 2nd Negotiated	
U Texas	Negotiated	[Browne]
DEC	Planned	

DEC Parallel Processing Conference in November

Livermore*para*VAX

THE PARA SUBROUTINES

- Mailboxes – Scheduling
- Global Sections – Shared Variables
- Event Flags – Barriers
- Locks – Critical Sections
- Binary Semaphores (Latches)
- Counting Semaphores (P & V)

Livermore *para*VAX

CONCLUSIONS

1. Description of the VAX 11/780-4

2. Research at LLNL

3. The Share Example

4. Experience and the Future

Livermore paraVAX

UTILITIES :

COSCOMPLINK *TASKNAME*

COMPILE AND LINK EACH USER TASK (PROGRAM) TO THE COSSHARE LIBRARY

COSSETUP *ROOTNAME [TASKNAME(S)]*

SET UP COMMAND FILES FOR ROOT TASK AND ALL NON ROOT TASKS

(needs to be done only once)

COSSUBMIT *ROOTNAME [AFTER_TIME]*

START UP ROOT TASK AT TIME=AFTER_TIME

IF AFTER_TIME NOT PRESENT, START UP NOW

COSCLEANUP *ROOTNAME*

REMOVES LEFT OVER BATCH JOBS AND

DELETES SHARED MEMORY ACCESS FOR THIS JOB

(needs to be done if job exits abnormally)

Livermore*para*VAX

SUBROUTINE DEFINITIONS:

CALL EVASGN (*EVENTDATA)*

ASSIGN, INITIALIZE EVENT

CALL EVREL(*EVENTDATA)*

IF THERE ARE WAITERS FOR EVENT --- ERROR

OTHERWISE DEASSIGN THIS EVENT

CALL EVPOST(*EVENTDATA)*

POST EVENT

CALL EVCLEAR (*EVENTDATA)*

CLEAR EVENT

CALL EVWAIT (*EVENTDATA)*

WAIT FOR EVENT

LOGICAL = EVTEST (*EVENTDATA)*

LOGICAL =.TRUE. IF EVENT WAS POSTED

LOGICAL =.FALSE. IF EVENT WAS CLEAR

SUBROUTINE DEFINITIONS:

CALL LOCKASGN (*LOCKDATA)*

ASSIGN, INITIALIZE LOCK

CALL LOCKREL (*LOCKDATA)*

IF THERE ARE LOCK WAITERS --- ERROR

OTHERWISE DEASSIGN THIS LOCK

CALL LOCKON(*LOCKDATA)*

IF LOCK IS BUSY WAIT, OTHERWISE GET LOCK

CALL LOCKOFF(*LOCKDATA)*

RELINQUISH THE LOCK

LOGICAL = LOCKTEST(*LOCKDATA)*

LOGICAL =.TRUE. IF LOCK WAS ON, RETURN

LOGICAL =.FALSE.IF LOCK WAS OFF,SETLOCK, RETURN

Livermore *para*VAX

SUBROUTINE DEFINITIONS:

CALL TSKSTART(*TASKARRAY, NAME [,LIST])*

STARTUP TASK

CALL TSKWAIT(*TASKARRAY)*

WAIT FOR TASK

LOGICAL = TSKTEST(*TASKARRAY)*

LOGICAL = .TRUE. IF TASK EXISTS

CALL TSKVALUE(*TASKVALUE)*

RETRIEVE TASKVALUE FOR THIS TASK

Livermore *para*VAX

SUBROUTINE DEFINITIONS:

CALL TASK_CLEANUP(*TASKARRAY [,LIST])*

if root task

DELETE SHARED MEMORY

if non root task

COPY TASKARRAY AND ARGUMENTS BACK

RELEASE LIBRARY SHARED MEMORY USED BY TASK

SUBROUTINE DEFINITIONS:

CALL SHAREDGLOBAL(*TASKARRAY*)

USE SHARED ADDRESSES FROM TASKARRAY TO MAP THE SHARED MEMORY.
ALSO MAP A BLOCK OF DATA TO BE USED BY THE SUBROUTINE LIBRARY FOR IMPLEMENTING EVENTS, LOCKS, ARGUMENT PASSING.
(used by root task only)

CALL GET_TASK_INFO(*TASKARRAY [,LIST])*

MAP LIBRARY SHARED MEMORY
USE LIBRARY SHARED MEMORY TO PASS ADDRESSES AND TASKARRAY (stored by tskstart subroutine)
USE SHARED ADDRESSES FROM TASKARRAY TO MAP THE USER SHARED MEMORY.
COPY TASKARRAY AND ARGUMENTS
(used by non root task only)

Livermore *para*VAX

PROPOSED COSSHARE LIBRARY

DATA DEFINITIONS:

TASKARRAY:	1ST QUADWORD	COUNT
	2ND QUADWORD	ENTRY #/ QUEUE NAME
	3RD QUADWORD	USER DEFINED VALUE
	4TH QUADWORD	BEG/END SHARED ADDR

NAME: TASK NAME (DESCRIPTOR)

LIST: ARGUMENT LIST (REFERENCE)

TASKVALUE: USER DEFINED, STORED AT 3RD QUADWORD OF TASKARRAY

LOCKDATA:	LONGWORD	REPRESENTING LOCK
EVENTDATA:	LONGWORD	REPRESENTING EVENT

Livermore*para*VAX

CRAY / VMS SUBROUTINES

*TASKING

- STARTUP TASK
- WAIT FOR TASK
- TEST IF TASK EXISTS

*BARRIER

- ASSIGN/RELEASE EVENT
- POST/CLEAR EVENT
- WAIT FOR EVENT
- TEST IF EVENT IS POSTED

*CRITICAL REGION

- ASSIGN/RELEASE LOCK
- TURN ON/OFF LOCK
- TEST IF LOCK IS ON

Livermore*para*VAX

VMS
MULTITASKING GROUP OF JOBS

*ROOT TASK - PROGRAM

*ALL OTHER TASKS - PROGRAMS

*LIBRARY OF SUBROUTINES

- MAP TASK SHARED DATA TO SHARED MEMORY
- PASS ARGUMENTS TO NON ROOT TASKS
- CLEANUP SHARED MEMORY BEFORE EXITING

*SET OF UTILITIES

- SET UP COMMAND FILES TO DEFINE TASK SHARED MEMORY ENVIRONMENT
- SUBMIT JOBS TO A BATCH QUEUE
- CLEANUP AFTER ABNORMAL EXIT

Livermore*para*VAX

CRAY (COS)

MULTITASKING

ON VMS ???

Livermore *para*VAX